

**METHOD AND SYSTEM FOR ENHANCED CONCURRENCY IN A COMPUTING  
ENVIRONMENT**

5

**BACKGROUND OF THE INVENTION**

CROSS REFERENCE TO RELATED APPLICATIONS

The present application claims priority from U.S.  
10 Provisional Patent Application Ser. No. 60/168,861, filed  
on December 2, 1999 by Herbert W. Sullivan and Clifford L.  
Hersh.

DESCRIPTION OF THE BACKGROUND ART

15 The present invention relates generally to  
multitasking computer systems, and more particularly, to  
multiprocessor systems and a method for increased  
concurrency in such systems.

Multiprocessor computing systems have been used for  
20 years to perform multiple tasks. A multiprocessing  
computing system may be defined as a computer assembled  
from a plurality of independently or semi-independently  
operating intelligent, i.e., processor-based, stations.  
The stations are generally interconnected for communication

by a communication bus. Each processor may perform a plurality of tasks with apparent concurrency. Distributing the system intelligence among a plurality of stations may improve the processing throughput of the computing system, allow the system to modularly and uniformly expand to meet increasing computing requirements, and permit greater efficiency and productivity through concurrency.

While multiprocessor computing systems are known, they are not scalable. The software of such systems is often highly dependent upon the configuration and characteristics of the system hardware. This means that the scalability of the multiprocessor is limited. Further, the dependency of the software on the hardware often causes a waste in processor resources.

Synchronizing or coordinating the tasks using a central processor to enhance concurrency of a plurality of tasks has been problematic in most multiprocessor systems. The prior patents of Sullivan, U.S. Patent Nos. 4,484,262, 4,707,781 and 5,438,680, the disclosures of which are incorporated herein by reference, addressed some issues relating to enhancing concurrency.

The Sullivan patents describe that concurrency can be enhanced by communicating between processors or tasks via commonly accessible memory to reduce conflicts within the

multiprocessor system. However, the disclosed techniques are primarily dependent on hardware, such as the processors.

Uninterruptible or atomic operations are operations  
5 that cannot be interrupted by another processor or by  
another thread on the same processor. Intel processors,  
for example, perform addition as an atomic machine  
instruction with extremely short locking implemented by a  
cache coherency mechanism. A LOCK prefix implements the  
10 atomic operation for a set of operations that do read-  
modify-write operations on a single memory address such as  
the LOCK XADD instruction (Exchange Add). The lock time of  
atomic instructions is generally very short.

Current technology suffers from the inability to add  
15 or subtract from a shared memory location without locking  
the location during the operation if it is necessary to  
keep the value within limits or maintain a valid  
transaction log. Using atomic operations, it is possible  
to add or subtract without locking which causes blocking,  
20 provided that there are no limits and a valid transaction  
log is not necessary.

Therefore, a need exists for a system and method for  
enhancing concurrent computation with shared resources.  
Further, a need exists for a software-based system that is

[illegible]

## SUMMARY OF THE INVENTION

In one aspect, the invention is directed to a method  
5 for operating a computer system. The computer includes at  
least one processor. The method includes establishing a  
plurality of memory units each having a corresponding  
memory location. A plurality of tasks running on the  
processor are executed, and the plurality of tasks are  
10 operable to share data. A plurality of lists for each  
memory location are defined, and at least one of said lists  
is locked if the data is invalid. An entry is inserted  
into said locked list corresponding to one of said tasks.  
The locked list is unlocked, and a determination is made if  
15 data is inputted in the memory location between the  
determining step and the unlocking step.

In another aspect, the invention is directed to a  
method for operating a computer system. The system  
includes at least one processor. The method includes  
20 establishing a plurality of memory units each having a  
corresponding memory location. A plurality of tasks are  
run on said processor, and the plurality of tasks are  
operable to share data. An entry corresponding to one of  
the tasks is inserted into one of the lists if the list is

unlocked. The method also includes determining if another of said lists is unlocked if said one list is locked.

In another aspect, the invention is directed to a method for synchronizing processes in a computer system.

5 The computer system includes at least one processor. The method includes establishing a plurality of memory units each having a corresponding memory location. A plurality of tasks running on the processor are executed, and the plurality of tasks are operable to share data located in  
10 the memory units. A plurality of lists for each memory location are defined. A list is locked if the data is not valid. An entry is inserted into the locked list corresponding to one of the tasks, and the list is unlocked. The entered task is suspended until valid data  
15 is found in the memory unit. The method also includes the steps of reading valid data and determining if other data appears in said memory location before said locking step and after said unlocking step. The other data is read if it appears in the memory unit.

20 Implementations of the invention include one or more of the following. The locking step further comprises activating selected other ones of said plurality of tasks that are entered on said locked lists. The plurality of lists may form a linked list. The plurality of lists may

be between four and eight. The method may also include the step of transferring the operation of said locked list when said locked list is locked by another one of said plurality of tasks.

5 In another aspect, the invention is directed to a computer system having enhanced concurrency that includes a plurality of processors. A plurality of tasks are run on the plurality of processors. The computer system may also include a plurality of memory units each having a  
10 corresponding memory location, and a plurality of lists corresponding to each of said memory locations. One of said tasks may be responsible for activating selected ones of said plurality of tasks contained on the same list as said one task.

15 In yet another aspect, the invention provides a system and method for adding and subtracting within limits without blocking. A thread comprises an operation including either addition of an addend or subtraction, which is performed as addition of a negative of the addend.  
20 The operation affects a shared actual value stored in an actual value register. Upper and lower limit registers store permissible upper and lower limits within which the result of the operation must fall. Addition and subtraction reservation registers, which are also shared

resources, store a reserved value after an addition to the actual value or after a subtraction from the actual value respectively.

The method includes getting the value of the addend.

- 5 If the operation is addition the value of the addend is positive, otherwise it is negative. A LOCK XADD operation using the addend is performed on an affected reservation register, i.e., the addition reservation register if the operation is addition or the subtraction reservation  
10 register if the operation is subtraction. The resulting value in the affected reservation register is then compared to the value of the limit registers. If the operation cannot succeed, the addend is added back to the affected reservation register in a LOCK XADD operation and a failure  
15 is reported.

- If the operation can succeed, in a LOCK XADD operation the addend is added to the value of the actual value register. Finally, in a LOCK XADD operation, the addend is added to the value of the unaffected reservation register  
20 which was not affected by the first atomic operation and a success is reported.

Aspects of the invention include a computer-implemented method for adding and subtracting within limits without blocking. Another aspect of the invention provides



a system having a central processing unit (CPU) and a memory which are configured to effect the method described above. Another aspect includes a computer program recorded on a computer readable medium for causing a computer to  
5 effect the method as described above.

The foregoing and other objects and advantages of the disclosed system and method will become apparent to those of ordinary skill in the art after having read the following detailed description of the preferred embodiments  
10 that are illustrated in the various drawing figures.

## BRIEF DESCRIPTION OF THE DRAWINGS

FIGs. 1A - 1D are flowcharts illustrating a preferred method of reading data from a mailbox.

FIGs. 2A - 2D are flowcharts illustrating a preferred  
5 method of inputting data in a mailbox.

FIG. 3 is a flowchart of a "one last look" method.

FIGs. 4 - 24 illustrate additional aspects of the invention.

FIG. 25 illustrates a portion of a conventional  
10 computer system, including a CPU and a memory, in which the present invention may be embodied.

FIG. 26 illustrates a system block diagram in accordance with a preferred embodiment of the invention.

FIG. 27 illustrates the overall process for  
15 implementing a method for adding and subtracting within limits without locking.

## DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

The present invention is directed to a system and  
5 method for enhancing concurrency in a multiprocessor or  
multitasking computer system. A multitasking computer  
system may be defined as a computer system that permits  
multiple threads of execution in a single execution space.  
A multiprocessing computer system may be defined as a  
10 computer system that allows multiple processing units to  
share data. Preferably, various tasks in the computer  
system communicate using commonly accessible "mailboxes."  
A mailbox is an area of memory that contains a number of  
states, and may be identified by a virtual address. When  
15 one task produces an item of valid data that may be desired  
by multiple other tasks, the task holding the valid data  
inputs it into a mailbox. Generally, other tasks read the  
valid data from the mailbox.

In accordance with a preferred method, the task that  
20 inputs the valid data into the mailbox is caused to notify  
other tasks addressing the mailbox that the data is  
contained therein. This means that there is no need for  
central coordination of mailbox accesses. Further, this  
minimizes busy waits for valid data. This permits tasks

and processors to allocate their resources more productively.

As stated above, the mailbox is a memory location. The mailbox includes a pointer to the address corresponding to valid data. Alternatively, a number, such as a string of zeros, is employed if the data is invalid. For simplicity, the preferred method will be described with reference to inputting or reading data to or from the mailbox. However, the data to be accessed can be located in the same memory system as the mailbox. Further, the term "task" refers to a set of instructions running on a process. Thus, the processor executes the task. Additionally, the terms "invalid" and "valid" refer to arbitrary logical status. In the preferred method, a task reads valid data and does not read invalid data. Other states may be used such as "true" and "false".

A plurality of lists of tasks wait for valid data associated with the mailbox. By coordinating several lists in accordance with the preferred method, conflicts in accessing data and delays in obtaining data can be minimized. This is because only a small number of instructions, for example, under 30, are required to read or place data to or from the mailbox. These functions occupy a small percentage of a task's activity time, such

as 5% or less. Accordingly, it has been found that a smaller number of lists can eliminate most conflicts. However, a list for each active task may be required to eliminate all conflicts. Any number of lists may be used.

- 5 For example, the number of lists may be between four and eight.

In the preferred method, the lists are a linked list. A linked list can employ locations in the same memory system as the mailboxes. The linked list includes

10 locations having suitable space for at least two pointers. The locations in the middle of the list may contain a forward pointer to the next location including the list, and a backward pointer to the preceding location including the list. Additionally, the top location on the list may

15 contain a pointer forward to the next location and a pointer backward to an illegal address, such as zero. The bottom location on the list can contain a pointer backward to the preceding location and a forward pointer to an

illegal address, such as zero. When an item, i.e., data,

20 is added to the list, the forward pointer corresponding to the previous bottom location is changed to correspond to the location of the added item. Also, the added item becomes the new bottom location. When the top item is taken from the list, the next item in the list replaces the

top item. Additionally, the backward pointer of the added item is changed to point to an illegal address, such as zero. When both pointers of an item in a mailbox are to an invalid address, the list is considered empty.

5

### **Reading Valid Data When Available**

FIGs 1A - 1D are flow charts showing a preferred method for reading data from a mailbox. Initially, data in a mailbox is read (step 101). In the preferred method, the mailbox may contain a pointer to the memory location containing the desired valid data or an illegal address corresponding to invalid data, such as zero. The mailbox is then addressed and interrogated to determine if the data is valid (step 102). At this stage, a mailbox may contain or is about to contain data associated with a program of a task seeking such data. If valid data is in the mailbox, it is read from the memory address indicated in the mailbox (step 104). The task may also read the data during step 102.

20 One aspect described herein is that the task performing steps 101 and 102 contacts other tasks waiting for valid data from that mailbox. This means that conflicts are avoided in the multiprocessor system because busy waits for valid data are minimized.

FIGs 1A - 1D illustrate lists L0 and L1 that represent tasks waiting to receive valid data from a mailbox, and FIGs. 2A - 2D shows lists L2 and L3 that identify tasks waiting to input valid data in the mailbox.

5       As shown in FIG. 1A, a task list is locked when it is manipulated by a task. This prevents access by other tasks to that list. After list L0 is locked (step 106), the task determines if list L0 was previously locked by another task (step 108). If list L0 was previously locked by another  
10 task prior to the current task executing steps 106 or 108, the prior task operates list L0. The current task then proceeds to check list L1 (step 122), as described below.

If list L0 was previously locked (step 108), the current task determines if list L0 is empty (step 110). If  
15 the list L0 is not empty, the current task then removes the first waiting task ("waiter") from the list (step 116). A waiter may be defined as a task that waits for valid data to be in a mailbox. In this configuration, a waiter may be in a suspended state, as described below. The current task  
20 then unlocks list L0 such that other tasks may access the list (step 118). Additionally, the current task causes the task associated with that waiter to execute a wake code (step 120). A wake code may be defined as a set of computer instructions that cause a task to activate from a

suspended mode. Then, the current task again locks list L0 (step 106). This process continues until each task on list L0 has been notified, and the list L0 is empty.

Alternatively, this process may continue until the list L0  
5 has been found locked. If this list is empty, list L0 is unlocked (step 112), and the task proceeds to check list L1 (step 122), as described below.

If list L0 was previously locked (step 108), the current task locks list L1 (step 122), and then checks if  
10 that list was previously locked (step 124). If list L1 was not previously locked at step 124, the current task determines if list L1 is empty (step 128). If list L1 is not empty, the current task removes the waiters from list L1 (step 132) and causes the waiters to execute their wake  
15 codes (step 134), as described above. This process repeats until list L1 is empty or the list L1 has been found locked. When list L1 is empty, it is unlocked (step 130). The task then continues to execute its remaining instructions as defined by the system (step 126). If list  
20 L1 was previously locked by another task prior to steps 122, 124 or 126, that task operates the locked list. The current task then executes step 126.



### Valid Data Not Available

If no valid data is found at the mailbox (step 102), list L0 is locked (step 140). The current task then determines if list L0 was previously locked (step 142). If  
5 the list was not previously locked, the current task is entered on list L0 (step 144). List L0 is then unlocked (step 146), and the current task checks for valid data (step 148). The current tasks then determines the validity of the data (step 150). This above process may be referred  
10 to as "one last look," as described below. In the preferred method, this configuration keeps two or more tasks, which are synchronized by passing data through the mailbox, from missing each other.

If the data is still not valid at step 150, the task  
15 is suspended (step 152). This releases computer system resources for other tasks, or the task can complete other system instructions. This means that the task does not busy wait or periodically check for valid data in the memory location. The current task will be activated by  
20 another task which has found valid data in the memory location, or by a task which previously inputted the valid data (step 154). When activated by another task, the current task proceeds to read the valid data from the mailbox (step 104). If the data is valid (step 150) and

list L0 was previously locked (steps 156 and 158), the current task is suspended until activated (steps 152 and 154). When activated by another task, the current task proceeds to read the valid data from the mailbox (step 104).

If the data was valid (step 150) and the current task determines that list L0 was not locked (step 156 and 158), the current task reads the data. This may occur as the current task determines the validity of the data. Then, the current task executes step 202.

If the current task is on the list L0 at step 202, the current task is removed from the list (step 204). The next waiting task is also removed from the list L0 (step 206), and list L0 is again unlocked (step 208). The task then causes the other task removed from the list to execute its wake code (step 210).

Next, in accordance with the preferred method, the current task determines if the list L0 was previously locked (steps 212 and 214). If so, the current task then executes other system instructions (step 220). If the list was not previously locked, the current task then determines if the list is empty (step 216).

If list L0 is not empty (step 216), the current task loops back to remove and notify the next task at the top of

the list (steps 206 - 216). If on any such loop, the list is found locked at steps 214 or 216, the current task continues to perform other system instructions (step 220). When the list is empty, it is unlocked (step 218) and the  
5 current task continues to perform other system instructions (step 220).

The current task may not be on the list L0 (step 202). This means that another task has removed the current task from the list and is operating to activate it. However,  
10 the current task also operates list L0. Thus, the current task must activate the other tasks on list L0. The current task may then suspend itself until activated by another task.

Steps 228 through 238 operate similar to steps 206  
15 through 218. However, when list L0 is found locked (step 234) or empty (step 236), the current task is suspended (step 240). When activated by another task (step 242), the current task proceeds to read the valid data from the mailbox (step 104).

20 Referring again to FIG. 1B, if list L0 was locked (step 142), the current task proceeds as follows. List L1 is locked and then the current task determines whether that list was previously locked (steps 180 and 182). If list L1 was previously locked, the current task returns to lock

list L0 (step 140), and continues to search for an unlocked list. This example provides two possible lists. However, more lists may be used. If the number of lists equals the number of tasks, no task will repeat this loop.

5        If list L1 was not previously locked (step 182), the current task becomes a waiter on list L1 (step 184), the list L1 is unlocked (step 186), and a "one last look" check is made for valid data (steps 188 and 190). If the data is not valid, the current task is suspended (step 152).

10       If the data is valid (step 190), then the current task reads the data as it confirms the validity of the data. List L1 is then locked (step 192), and a check is made whether it was previously locked (step 194). If list L1 was previously locked (step 194), list L1 is unlocked (step 15  
15    196), and the current task is suspended (step 152).

      If the data was valid and it is determined that list L1 was not locked, at steps 190, 194, and 196, respectively, then the current task proceeds (step 252).

      FIG. 1D shows that if the current task is on list L1  
20    (step 252) that task is removed from the list (step 254), and also removes the next waiting task from the list L1 (step 256). The task then unlocks list L1 again (step 258). The task then causes the other task removed from the list to execute its wake code (step 260).

Next, the current task determines whether the list L1 was previously locked (steps 262 and 264). If it was not previously locked, the current task determines whether the list is empty (step 266).

5        If list L1 is not empty (step 266), the current task loops back to activates task at the top of the list (steps 256 through 266). If on any such loop, the list is found locked at steps 264 or 266, the current task continues step (270). When the list is empty, it is unlocked (step 261),  
10      and the current task continues to execute other system instructions (step 270).

         If the current task is not found on the list L1 (step 252), this means that some other task has taken the current task off the list and is attempting to activate it. The  
15      current task also operates list L1. Thus, this task must activate the other tasks on list L1 before suspending itself. The current task can then be activated by another task. The method of steps 278 through 288 is similar to steps 256 through 268 described above. However, when list  
20      L1 is found locked (step 284) or empty (step 286), the current task is suspended (step 290). When activated by another task (step 292), the current task proceeds to read the valid data from the mailbox at step 104.

This preferred method of finding an available list is simple. Further, a task should not be significantly delayed in finding an unlocked list. Thus, enhanced concurrency can be achieved over prior systems.

5

	1970	1971	1972	1973	1974	1975	1976	1977	1978	1979	1980	1981	1982	1983	1984	1985	1986	1987	1988	1989	1990	1991	1992	1993	1994	1995	1996	1997	1998	1999	2000	2001	2002	2003	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013	2014	2015	2016	2017	2018	2019	2020	2021	2022	2023	2024	2025	2026	2027	2028	2029	2030	2031	2032	2033	2034	2035	2036	2037	2038	2039	2040	2041	2042	2043	2044	2045	2046	2047	2048	2049	2050	2051	2052	2053	2054	2055	2056	2057	2058	2059	2060	2061	2062	2063	2064	2065	2066	2067	2068	2069	2070	2071	2072	2073	2074	2075	2076	2077	2078	2079	2080	2081	2082	2083	2084	2085	2086	2087	2088	2089	2090	2091	2092	2093	2094	2095	2096	2097	2098	2099	2100	2101	2102	2103	2104	2105	2106	2107	2108	2109	2110	2111	2112	2113	2114	2115	2116	2117	2118	2119	2120	2121	2122	2123	2124	2125	2126	2127	2128	2129	2130	2131	2132	2133	2134	2135	2136	2137	2138	2139	2140	2141	2142	2143	2144	2145	2146	2147	2148	2149	2150	2151	2152	2153	2154	2155	2156	2157	2158	2159	2160	2161	2162	2163	2164	2165	2166	2167	2168	2169	2170	2171	2172	2173	2174	2175	2176	2177	2178	2179	2180	2181	2182	2183	2184	2185	2186	2187	2188	2189	2190	2191	2192	2193	2194	2195	2196	2197	2198	2199	2200	2201	2202	2203	2204	2205	2206	2207	2208	2209	2210	2211	2212	2213	2214	2215	2216	2217	2218	2219	2220	2221	2222	2223	2224	2225	2226	2227	2228	2229	2230	2231	2232	2233	2234	2235	2236	2237	2238	2239	2240	2241	2242	2243	2244	2245	2246	2247	2248	2249	2250	2251	2252	2253	2254	2255	2256	2257	2258	2259	2260	2261	2262	2263	2264	2265	2266	2267	2268	2269	2270	2271	2272	2273	2274	2275	2276	2277	2278	2279	2280	2281	2282	2283	2284	2285	2286	2287	2288	2289	2290	2291	2292	2293	2294	2295	2296	2297	2298	2299	2300	2301	2302	2303	2304	2305	2306	2307	2308	2309	2310	2311	2312	2313	2314	2315	2316	2317	2318	2319	2320	2321	2322	2323	2324	2325	2326	2327	2328	2329	2330	2331	2332	2333	2334	2335	2336	2337	2338	2339	2340	2341	2342	2343	2344	2345	2346	2347	2348	2349	2350	2351	2352	2353	2354	2355	2356	2357	2358	2359	2360	2361	2362	2363	2364	2365	2366	2367	2368	2369	2370	2371	2372	2373	2374	2375	2376	2377	2378	2379	2380	2381	2382	2383	2384	2385	2386	2387	2388	2389	2390	2391	2392	2393	2394	2395	2396	2397	2398	2399	2400	2401	2402	2403	2404	2405	2406	2407	2408	2409	2410	2411	2412	2413	2414	2415	2416	2417	2418	2419	2420	2421	2422	2
--	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	---

### Inputting Valid Data

FIGs. 2A - 2D are flow charts showing the input of data into a mailbox in accordance with the preferred method. Initially, data is read in a mailbox (step 301). The mailbox is then addressed and interrogated to determine whether the data is valid (step 302). If valid data is not already in that mailbox, the mailbox is marked as not containing valid data. Next, the valid data is placed in the mailbox (step 304).

In accordance with the preferred method, lists L2 and L3 identify tasks waiting to input valid data in the mailbox. This includes a current task waiting to input data in the mailbox. The current task determines whether list L2 was previously locked (step 304). If so, that list remains locked (step 306). If list L2 was not previously locked the current task determines whether list L2 is empty (step 310). If the list L2 is not empty, then the current task removes the first waiter from that list (step 316).

Also, the current task unlocks list L2 so that other tasks may access it (step 318), and causes the task associated with that waiter to execute its wake code (step 320).

Then, the current task locks list L2 again (step 306). The above process continues from step 306 until all tasks on

list L2 seeking the valid data from the mailbox have been activated. Alternatively, this process continues until the list L2 is empty, or the list L2 has been found locked. When list L2 is found empty, it is unlocked (step 312), and  
5 the current task proceeds to list L3 (step 322).

If list L2 was previously locked (step 301) or list L2 was found empty (step 310) and then unlocked (step 312), the current task then locks list L3 (step 322). Then, the current task checks if list L3 was previously locked (step  
10 324). If list L3 was not previously locked, the current task determines whether list L3 is empty (step 328). If list L3 is not empty, the current task proceeds to remove the waiters from list L3 (steps 332 and 334) and causes them to execute their wake codes (step 306). This process  
15 repeats by looping back to step 322. The process may continue until list L3 is empty or has been found locked. When list L3 is empty, it is unlocked and the current task continues to execute other system instructions (steps 330 and 338). If list L3 is found to have been previously  
20 locked at steps 322 or 324, this indicates activity by another task. The current task then executes other system activities (step 338). The current task also defers to the other task to activate the other tasks listed on list L3.



## Valid Data Found in Mailbox

5 If the current task determines that there is valid data at the memory location (step 302), the list L2 is locked (step 304). The current task also checks whether the list was previously locked (step 342). If list L2 was not previously locked, the current task is inserted onto list L2 (step 344). List L2 is then unlocked (step 346) and a "one last look" check is made for valid data (steps 348 and 350), as described below. If the data is still  
10 invalid, the current task is suspended (step 352). The current task is then activated by another task that has found or inputted valid data in the memory location (step 354).

15 If the data in the mailbox is valid and list L2 was previously locked (steps 356 and 358), the current task is suspended until activated (steps 352 and 354). When activated by another task, the current task proceeds to input valid data into the mailbox (step 304).

20 If the data was not valid (step 350) and list L2 was not previously locked (steps 356 and 368), the current task continues to execute other system commands (step 402).

If the current task is on list L2 (step 402) the current task removes itself from that list (step 404). Also, the current task removes the next waiting task from

the list L2 (step 406) and unlocks list L2 again (step 408). The current task then causes the other task to be removed from the list to execute its wake code (step 410).

In accordance with the preferred method, the current  
5 task determines whether the list L2 was previously locked (steps 412 and 414). If so, the current task proceeds to input valid data in the mailbox (step 304). Otherwise, the current task determines whether the list is empty (step 416).

10 If list L2 is not empty (step 416), the current task loops back to remove and activate the next task at the top of the list (steps 406 - 416). If on any such loop, the list is locked at steps 414 or 416, the current task inputs valid data in the mailbox (step 304). When the list is  
15 empty, it is unlocked (step 418). The current task then inputs valid data in the mailbox (step 304).

If the task is not on the list L2 (step 402), the current task may be removed from the list and is being activated by another task. The current task also operates  
20 list L2. Thus, the current task must activate the other tasks which are on list L2 before suspending itself. The current task may then be activated by another task. The method of steps 428 through 438 is similar to steps 406-418. However, when list L2 is found previously locked

(step 434) or empty (step 436), the current task is suspended (step 440). When activated by another task (step 442), the current task proceeds to input valid data in the mailbox (step 304).

5        If list L2 was previously locked at step 342, the current task locks list L3 (step 380) and then determines whether that list was previously locked (step 382). If list L3 was previously locked, the current task loops back to attempt to access list L2 (step 340), and continues to  
10   find an unlocked list. This example provides two possible lists. However, more than two lists may be employed.

      If list L3 was not previously locked (step 382), the current task is inserted on list L3 as a waiter (step 384). The list L3 is also unlocked (step 386), and a "one last  
15   look" check is made for valid data (steps 388 and 390), as described below. If the data is valid, the current task is suspended (step 352).

      If the data is not valid (step 390), list L3 is locked, (step 392) and a check is made whether list L3 was  
20   previously locked (step 394). If list L3 was previously locked, list L3 is unlocked (step 396), and the current task is suspended (step 352). If the data was not valid and the list L3 was not previously locked (steps 390, 394, and 396), the current task proceeds to step 452.

If the current task is on list L3 (step 452), it is removed from that list (step 454). Also, the next waiting task from the list L3 is removed (step 456), and the current task unlocks list L3 again (step 458). The current  
5 task then causes the other task removed from the list to execute its wake code (step 460).

The current task then determines whether the list L3 was previously locked (steps 462 and 464). If it was not previously locked, the current task determines whether the  
10 list is empty (step 466).

If list L3 is not empty, the current task loops back to remove and activate the next task at the top of the list (steps 456-466). If on any such loop, the list is locked (steps 464 and 466), the current task inputs valid data in  
15 the mailbox (steps 304). When the list is empty, it is unlocked (step 468), and the current task determines if valid data is inputted in the mailbox (step 304).

If the current task does not find itself on list L3 (step 452), this may be because some other task has taken  
20 the task off the list and is trying to activate it. The current task, however, operates list L3. Thus, the current task must activate other task on list L3 before suspending itself. The current task then may be activated by another task. The method of steps 478 through 488 is similar to

steps 456 through 468. However, when list L3 is found previously locked (step 484) or empty (step 486), the current task is suspended (step 490). When activated by another task (step 492), the current task proceeds to input  
5 valid data in the mailbox (step 304).

### One Last Look

The term "one last look" refers to the situation where a task that has placed itself on a list for valid data from  
10 a mailbox receives that data. The preferred method allows tasks addressing the same mailbox to be coordinated. This means that a current task inputting valid data in a mailbox will activate another task that was entered onto a wait  
list, while the data was placed in the mailbox by the  
15 current task. Accordingly, a current task may not input valid data in the mailbox, while another task is inserted on a list to receive that data. The preferred method avoids the potentially infinite period of time that a task may sleep until another task activates it. Thus, delays in  
20 system processing time may be minimized.

FIG. 3 shows that a task first looks at the data in the selected mailbox and determines whether it is valid data (steps 548 and 550). If the data is valid, it reads

the data and continues with other system activities (steps 551 and 553).

At this stage, if the data is not valid, the task is inserted onto a wait list (step 555). Then, the "one last  
5 look" determines if the data is valid (steps 557 and 559). This occurs because, while the task that may have located an unlocked list, locked it, placed itself on the list and unlocked the list again, another task may have put valid data in the mailbox. Without this preferred "one last  
10 look" method, activating the inserted task may be delayed until another task seeks valid data from the mailbox.

If the data is not valid at that time, the task will suspend until activated by another task, such as the next task inputting valid data in the mailbox (steps 552 and  
15 554). This method corresponds generally to steps 352 and 354.

If the data is valid at that time (step 559), the task can take itself off the list (step 561). Alternatively, the task may read the data and continue its other system  
20 activities. This means that the task may assume that the data was initially found in the mailbox (steps 563 and 553). A number of embodiments of the present invention have been described. Nevertheless, it will be understood that various modifications may be made without departing

from the spirit and scope of the invention. For example,  
the number of lists may be increased to accommodate  
additional tasks. Accordingly, it is to be understood that  
the invention is not to be limited by the specific  
5 illustrated embodiments, but only by the scope of the  
appended claims.

#### Method For Synchronization of Multiple Asynchronous Threads

Without Spin Locks,

10

Busy Waits or Polling

Current technology does not possess the ability to  
synchronize multiple asynchronous threads without employing  
busy waits, or polling. This causes a waste of processor  
resources and limits the scalability of multi processors,  
15 because the likelihood of busy waits and the time consuming  
polling increases with the number of active threads. In  
addition, it is inefficient with current technology to  
multi thread programs in which a large number of elements  
20 effect the state and a large number of elements take the  
state as input (such as simulations).

An unlimited number of threads may interact without  
ever busy waiting or polling in an embodiment. It is  
possible to write programs which can execute efficiently on

uni processors and arbitrarily large multi processors  
without modification.

5 A thread of execution may be defined as a sequence of  
instructions which are parsed out of memory and executed by  
the processing hardware and whose state is maintained on a  
single stack. A mailbox may be defined as an area of  
memory which contains at least two states (usually called  
the valid state and the invalid state). A multilist may be  
defined as a structure composed of two or more lists which  
10 logically act as one list.

Associated with this mailbox is at least one structure  
which can contain references to threads of execution which  
have been suspended pending on of the permitted states of  
the mailbox. One implementation of this structure is a  
15 linked list. Threads will contain instructions which  
reference this mailbox. One implementation of these  
instructions is as methods of a mailbox object. This  
instruction, when executed, tests the state of the mailbox,  
and if the state is one which permits the completion of the  
20 instruction (called the valid state), completes the  
instruction. If the state of the mailbox is invalid, the  
thread is placed on a list associated with the invalid  
state and the thread is suspended (this method is  
appropriate for both user level control and system level



control. In this example only user level control will be explained. In user level implementations, the suspension is at user level and does not involve a system call).

There is another kind of instruction which changes the

5 state of the mailbox (Instructions may both wait on a state and change the state.) from invalid to valid. When this instruction is executed by any thread, all threads on the newly valid wait list cease to be suspended and are made runnable. The preferred methods may include many  
10 implementations of system level synchronization and event mechanisms.

A difficulty with any mechanism which suspends and resumes threads of execution is that a list must exist somewhere of the threads to be resumed and that more than  
15 one thread cannot be altering the list at any one time. This means that the list must be locked when it order is being altered. If another thread requires access to the list, it must wait its turn (busy wait). It does not help to suspend the thread because that just requires another  
20 identical "wait list". It also does not help to pass the problem to a single thread because that also creates an identical list for the attention of the single thread (It also limits scalability by creating a limit on the number of processors which can perform a particular function).

One embodiment includes several parts and solves the general case of access contention. Applications include not only synchronization, but also the general case of concurrent programming as exemplified by simulations, and  
5 by the non blocking access to databases by multiple read and write operations.

By using multiple lists (or queues etc.) to record the waiting threads and allowing any thread which finds its first choice of lists locked to use another one (and  
10 another if that one is locked), the chances of busy waiting can be made arbitrarily small for any thread trying to get itself on the list. The chance of busy waiting can even fall to zero if the number of lists is equal to the number of threads needing access. This may not be necessary  
15 because the probability of busy waiting declines faster than exponentially and approaches zero (becomes an insignificant barrier to scalability) for very small numbers of lists.

Accordingly, a solution to the input problem may be to  
20 use multiple lists.

If thread one finds the mailbox data invalid, and goes to put itself on a list, but thread two, after thread one has checked the data, makes the data valid and resumes all the threads on the list before thread one is even on the

wait list, thread one will wait forever. This is a  
synchronization failure. The solution to this  
synchronization problem is that thread one must check back  
("one last look") on the mailbox data after it has placed  
5 itself on the list and released the list. If it finds that  
the data is valid, it must remove itself from the list and  
continue. Now it is impossible for the synchronization to  
fail.

To reduce or eliminate the chance that a busy wait  
10 will occur because the reader and writer of synchronizing  
data contend for the same list, since they must share  
access for the synchronization to have meaning, access  
contention may occur. The solution to this access  
contention is as follows:

- 15
- A: A thread which is looking for its valid state,  
checks the mailbox data and if it is valid, returns  
that data and continues.
- 20 B: If that thread finds the data invalid, it finds an  
unlocked list (as above in part one) and locks it.  
It then places itself on the list, unlocks the list,  
and takes one last look at the mailbox data. If the  
data is still invalid the thread suspends itself.

C: If the data has become valid, the thread tries to  
lock the list on which it has placed itself. If the  
attempted lock fails, the thread suspends itself, it  
5 will be reawakened by whomever has the list locked.

D: If the lock succeeds, the thread removes itself from  
the list and causes all other members of the list to  
resume. The thread then continues.

10

1: A thread which is creating a valid state for the  
mailbox, writes data indicating that state to the  
mailbox data area and then attempts to lock the  
first list of the multilist. If the lock succeeds,  
15 the threads waiting on the list are caused to  
resume. The thread which wrote the valid data then  
repeats (1 or 2) the procedure on the next list of  
the multilist.

20

2: If the lock attempt fails, the list is skipped, and  
the one locking the list will cause all of the  
threads on the list to resume (see D). The thread  
which wrote the valid data then repeats (1 or 2) the  
procedure on the next list of the multilist.

3: When the last list of the multilist is completed,  
the thread which wrote the data will continue.

5        This mechanism may guarantee that the synchronization  
mechanism will always be successful and that it will never  
busy wait. This introduces a fixed amount of overhead to  
the communication and coordination of threads which does  
not increase with the size of the program or the number of  
10 processors. This generally defines scalability.

It may be useful to permit synchronization on any  
Boolean. A Boolean "not" can be implemented by having a  
thread have as its valid state of a particular mailbox the  
complement of another thread's valid state (or zero and not  
15 zero etc.). A Boolean "or" can be implemented by having  
two or more threads write valid data to the same mailbox.  
A Boolean "and" can be implemented by having a single  
thread read two or more mailboxes in succession. With  
'not', "and" and "or", all Booleans can be encoded. This  
20 provides for the very useful control structure, "whenever  
(general Boolean condition), do (statement)" without  
polling.

One problem is making simulations and other general  
programs concurrent. A method, analogous to the

synchronization mechanism, makes writing parallelizable simulation programs (and other inherently concurrent programs) easier. In this method:

5       A: The data is replaced by a list is associated with  
consecutive integers and which can be added to on  
one end only (an ordered list).

10       B: Multiple wait lists, ordered by the integer position  
being waited on, replace the multilist.

15       C: A request for data passes an integer and returns the  
data associated with that integer if the integer is  
lower than or equal to the end of the list. If it  
is higher than the number (any compare operator  
which divides the list at a point will do), the  
thread is placed on a wait list and the mechanism  
described above is used to resume it, except that  
the test of validity is whether the integer last  
20       written is equal to greater than the number  
requested.

Another problem making simulations and other general programs concurrent is that there is not always an obvious way to determine whether all segments of the program which

might influence the inputs to a particular segment have  
executed. A method, analogous to the synchronization  
mechanism, makes writing parallelizable simulation programs  
(and other inherently concurrent programs) easier. In this  
5 method:

A: A map is made of the program, which models a space,  
in which regions may be defined as exclusively permitting  
the interaction of components in a particular circumstance.  
For instance, in a driving simulation, automobiles would  
10 only effect one another if they were in identical or  
adjacent regions of the street map. The objects which move  
in the model may be attached to an arbitrary number of  
regions, called interaction regions, by enrolling in lists  
such as described in continuation 2. Each region is  
15 required to report the transfers of object to its neighbors  
(a neighbor is a region which may accept an object from the  
region to which it is a neighbor). When the number of  
reports equals the number of regions for a given integer  
value, that integer value is current (able to execute  
20 knowing all inputs are present). A preferred  
implementation is to count with a fetch-decrement  
instruction and pass objects from region to region by using  
a list mail box.

## Lock Free Record Management

There are two reasons for lock records. One is to keep it from being altered in inconsistent ways and the other is to protect it in some way. The first reason can be eliminated. Currently, locking dominates the cost of computing. Existing technology can do nothing to reduce this cost.

A mechanism for ordering operations to provide the same result as if there were locks without the need to block processes or threads is described. If the concurrency is high enough to overcome the inherent latency of access, a high percentage of optimal computing efficiency can be achieved. Accordingly, the number of necessary locks can be reduced.

A thread of execution may be defined as a sequence of instructions which are parsed out of memory and executed by the processing hardware and whose states is maintained on a single stack. A mailbox may be defined as area of memory which contains at least two states (usually called the valid state and the invalid state). A multilist can be defined as a structure composed of two or more lists which logically act as one list. Associated with this mailbox is at least one structure which can contain references to



threads of execution which have been suspended pending one of the permitted states of the mailbox. One implementation of this structure is a linked list. Threads may contain instructions which reference this mailbox. One

5 implementation of these instructions is a methods of a mailbox object. An instruction, when executed, tests the state of the mailbox, and if the state is one which permits the completion of the instruction (called the valid state), completes the instruction. If the state of the mailbox is  
10 invalid, the thread is placed on a list associated with the invalid state and the thread is suspended (This method is appropriate for both user level control and system level control. In this example only user level control will be explained. In user level implementations, the suspension  
15 is at user level and does not involve a system call).

There is another kind of instruction, which changes the state of the mailbox (Instructions may both wait on a state and change the state.) from invalid to valid. When this instruction is executed by any thread, all threads on the  
20 newly valid wait list cease to be suspended and are made runnable. Many implementations of system level synchronization and event mechanisms can be included. A difficulty with any mechanism, which suspends and resumes threads of execution is that a list must exist somewhere of

the threads to be resumed and that more than one thread cannot be altering the list at any one time. This means that the list must be locked when its order is being altered. If another thread requires access to the list, it must wait its turn (busy wait). It does not help to suspend the thread because that just requires another identical "wait list". It also does not help to pass the problem to a single thread because that also creates an identical list for the attention of the single thread (It also limits scalability by creating a limit on the number of processors which can perform a particular function). Applications include not only synchronization, but also the general case of concurrent programming as exemplified by simulations, and by the non blocking access to databases by multiple read and write operations. Generally, the general case of access contention can be solved.

A read-modify-write operation that may be added (such as LOCK XADD on the Pentium) and can be used to distribute locally unique consecutive numbers. Operations that must be performed in order are assigned numbers such that the operation that must be done first has a lower number. Each operator that can access the database can place commands in a list sorted by number. So long as each operation executes on individual records in the correct order, the

result must be the same. That this must be true is obvious from the fact that all locks do is guarantee some instruction will occur before some others.

A necessary instruction may not be delivered to the operation until an operation that needed to occur after it had already been executed in the above read-modify-write operation. A synchronizing mechanism should exist.

Instead of locks, a block delivery system can be used.

Instructions that must be executed in order are delivered in blocks. The synchronizing mechanism can be mailboxes.

#### Method for Adding and Subtracting within Limits without Blocking

Current technology suffers from the inability to add or subtract from a shared memory location without locking the location during the operation if it is necessary to keep the value within limits or maintain a valid transaction log. Using the uninterruptible operations common to most general purpose CPUs, it is possible to add or subtract without blocking, provided that there are no limits and a valid transaction log is not necessary.

It has been found that an unlimited number of threads to add and subtract from a single shared memory location

without blocking, overflow, or underflow. Further, the transaction log can always be valid so that error recovery is possible.

A thread of execution may be defined as a sequence of  
5 instructions which are parsed out of memory and executed by  
the processing hardware and whose state is maintained on a  
single stack. A shared value can be characterized as an  
area of memory that is accessible from at least two threads  
of execution. An uninterruptible instruction may be an  
10 instruction that reads an area of memory and does not  
permit another thread of execution to read or write that  
same area of memory until the instruction has completed. A  
limit can be defined as a value that the contents of a  
memory location must not pass (such as zero for inventory  
15 items or the end of allocated memory space for a pointer).

A single memory location can be included that may be  
accessed only by an interruptible add or subtract operation  
such as LOCK XADD (exchange and add) on the Pentium or a  
nondestructive operation such as a read. Such a device may  
20 safely permit many threads to add or subtract, but it is  
impossible to determine whether incomplete transactions  
have occurred or to execute instructions, such as extended  
precision, that require synchronization.

One more memory location (for a total of two) may be added that can be accessed by uninterruptible instructions only, whatever limits exist (limits may exist in one direction only) and a transaction log. Now an addition or subtraction is accomplished by adding (a negative number for subtraction) to one of the locations (each thread must use the two locations in the same order), if the operation does not place the total on the wrong side of the limits, then the operation is recorded in the transaction log and then added to the second memory location. If the total violates the limit (is smaller than the lower limit or larger than the upper limit) the number is subtracted from the first number. The failure may be recorded in a transaction log. Generally, this is an implementation detail. This device can guarantee a correct result, and that the total includes only valid operations, but it leaves a significant chance that a valid transaction will not occur.

A preferred embodiment of the invention is practiced in the context of a conventional personal computer such as an IBM compatible personal computer, an Apple Macintosh computer or a UNIX-based workstation. FIG. 25 shows a representative hardware configuration of a computer 2500 including a central processing unit (CPU)

2510, such as a microprocessor, and a number of other units interconnected via a system bus 2520. The computer 2500 may also include Read Only Memory (ROM) 2540, Random Access Memory (RAM) 2550, Non-Volatile Memory 2560, input devices 5 2570 (such as a keyboard, mouse, microphone, and touch screen) and output devices 2580 (such as a display screen, printer, and speaker) coupled to the system bus 2520. A Network Connection 2590 may be provided for connecting computer 2500 to a communication network (not shown) such 10 as an intranet or the Internet. The coomputer 2500 typically operates under control of an operating system such as the Microsoft Windows NT or Windows/98 OS, IBM OS/2, MAC OS, or UNIX operating system. Those skilled in the art will appreciate that the disclosed system and 15 method can also be implemented on platforms and operating systems other than those mentioned.

With reference to FIG. 26, a system for adding within limits without blocking is shown generally designated 2600. The system includes an actual value register 2610, a 20 subtraction reservation register 2620, an addition reservation register 2630, a lower limit register 2640, and a upper limit register 2650. RAM 2550 may include the registers 2610, 2620, 2630, 2640, and 2650.



If the operation is addition, the value of the addend is positive, otherwise it is negative.

In a step 2720 a LOCK XADD operation is performed upon the value stored in an affected reservation register using  
5 the addend. If the operation is addition, the affected reservation register is the addition reservation register 2630, otherwise it is the subtraction reservation register 2620. Thus in the exemplary case, the addend -3 is added to the subtraction reservation register 2620, resulting in  
10 a value of 4 in the subtraction reservation register 2620.

In a step 2730, the value of the affected reservation register is compared to the value of the limit registers 2540 and 2550. In a step 2740, it is determined if the operation can succeed within the limits. In the exemplary  
15 case, the value of the subtraction reservation register 2620 is 4, which is greater than or equal to the lower limit of 3 stored in lower limit register 2540 and less than or equal to the upper limit of 10 stored in the upper limit register 2650. Therefore the operation can succeed  
20 within the limits.

If the operation cannot succeed, in a step 2750, the value of the affected reservation register is restored by performing a LOCK XADD using the negative of the value. Thus in the example, 3 is added back to the value of 4



stored in the subtraction reservation register 2620 to restore the initial value of 7 stored in the subtraction reservation register 2620. Finally, a failure is reported in a step 2760 and the process ends.

5        If the operation can succeed, in a LOCK XADD operation 2770, the addend is added to the value stored in the actual value register 2610 and written to the actual value register 2610. In a step 2780, in a LOCK XADD operation, the addend is added to the reservation register unaffected  
10 by the first LOCK XADD operation of step 2720. In the exemplary case, the unaffected reservation register is the addition reservation register 2660 and thus -3 is added to 7 and loaded into the addition reservation register 2660. Finally, in a step 2790, a success is reported and the  
15 process ends.

A wait list with an uninterruptible counter may be added. Now, a thread of execution checks that the value could be added to the second memory location within the limit, performs an uninterruptible add on the first  
20 location as above, and if it succeeds, then it proceeds as described above. If it fails the test against the limit, the second value I checked. If the operation would succeed on the second value (the operation to test the second value cannot alter it), the thread calculates the chances of

success of each member of the wait list, in order, ignoring all members marked as causing overflow and marking all members that cause overflow. If the value of the second memory location plus the members of the list make it impossible to add the current value in without overflow, the value is subtracted back out of the first memory location and an overflow failure is reported. If the value can be succeed, the thread adds one to the counter and places itself in the location that the counter indicates (such as value times address length plus offset) with space for a read modify write operation. The thread then suspends for a time long enough to allow the last thread to successfully complete an operation to have a chance to mark the reserved location to indicate that the thread should continue. This time should be short enough to be otherwise benign. When the thread is reactivated it marks the space reserved for read-modify-write operation that it is done, if it has not been otherwise marked, then it subtracts the value it added from the first memory location. The last thread then marks members of the list that best exhaust the remaining space to the limit (application depended) and goes on. This device guarantees that an optimal result will be achieved without significant overhead and without much blocking, but it does not guarantee a unique last

thread. It is possible that two threads will try to finish allocating the remaining value to the limit, and it is possible that some threads waiting to be serviced will be missed.

- 5 "One last look" and synchronization configurations may also be added. For implementation, two locations are added to hold semaphores. A semaphore may be defined as a memory location only written to by uninterruptible instructions. One case may be considered. When a value greater than one
- 10 is added to the first total and causes the limit to be exceeded, but the second total is or has some room left before hitting the limit. There must be a semaphore in a known state (say reset). The thread that set the first value past the limit changes the semaphore (say sets it).
- 15 It then subtracts its value from the first memory location. If the result is within limits, the thread then checks the counter. If the counter is zero, it resets the semaphore. If the counter is non-zero, it adds the value of each thread on the list to the second total. Because the
- 20 semaphore blocks other writing done after the semaphore, and the first counter was known to be in bounds after the semaphore was set, there can be no case of overflow. If the first counter is out of bounds, values on the list are subtracted from the first counter until the counter is in

bounds, then proceed as before. Threads arriving at the device check the semaphore. If it is reset, they proceed as in part three. If it is set, the value of the second memory location is checked. If it does not allow the transaction, a failure is returned. If the second memory location has a value that allows the transaction to complete, then the value is added to the first location and the value to be added is placed on the list and the thread is paused. When it awakens, if it marked as added, it returns success. If it is marked failed, it returns failure. If it is unmarked (meaning that is miss-connected with the thread that set the semaphore), then it subtracts its value from the first memory location and begins again. Because contention occurs only when a large value exceeds the limit because other values were processed between the check of the second value and the addition to the first value and smaller values are waiting to be processed, contention almost never occurs. Locks are otherwise unnecessary. An incorrect result may never occur.

## Lock Free Lists

With current methods, it is necessary to lock a list to add or delete a member. It has been found that elements  
5 can be placed and removed without locks.

The lists may have many of the useful properties of queues without locks. These devices may also replace linked lists without locks.

The device may include a memory location and a  
10 contiguous block of memory. The memory location and the block are initialized a known value (like zero). Any thread of execution that would add to the list creates an array in memory and then adds one to the memory location with an uninterruptible instruction (such as XADD on the  
15 Pentium). From the value returned by the uninterruptible instruction, an address in the contiguous block of memory is computed. A pointer may be placed there pointing to the array just created. A thread can remove an element from the list by reading down the list and exchanging the first  
20 nonzero element with zero. This will be a pointer to an element of the list. If the thread trying to remove an element reads past the current write pointer, the list is empty.

The contiguous block of memory may be divided into multiple parts. Each part may have a memory location for a down counter and for a write location pointer. Each also may have a lowest unread pointer. In the rare case that a  
5 thread is suspended after it has gotten a write location and before it has written, locations after it on the list will be removed before it. The lowest unread pointer that allows the search to start as far down the list as possible without missing anything. When the down counter, which is  
10 decremented on each removal, is zero, the list may be reused. This permits the list to act like a queue, but without the locking of the head and tail pointers. (it scales).

A possible sequence may be implemented. The  
15 initializing process sets all of the memory to some known set of values that cannot be valid data. The down counters are set to the number of elements in a list section. The memory locations are set to the head of their respective sections of the list. All of the memory locations that  
20 point to the head of a contiguous portion of memory are placed in a list. The top of the list is pointed to in a master location.

The writing process increments the first section pointer by the block size. The XADD instruction returns

the old value of the pointer, which is the top of the list. If writes into this location the pointer that it wishes to place in the queue. The process continues until the pointer equals the endpoint of the section. Then the  
5 writing process finds the first section that is empty (down-counter equals zero), sets the down-counter to the section length and places the address of the section pointer in the master pointer. The next writer uses the next block referred to by the counter.

10 The removing process reads down the section from the top until it finds a non zero element. It exchanges that element with zero. If a non-zero value is returned, that element is removed and the responsibility of the remover. Otherwise, it continues to scan down until it succeeds in  
15 getting by uninterruptible exchange a non-zero value. The removing process then decrements the down-counter associated with the section in question and if the down counter equals zero, marks the section empty and available for re-use.

20 Several embodiments are specifically illustrated and/or described herein. However, it will be appreciated that modifications and variations are covered by the above teachings and within the scope of the appended claims

11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223 224 225 226 227 228 229 230 231 232 233 234 235 236 237 238 239 240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255 256 257 258 259 260 261 262 263 264 265 266 267 268 269 270 271 272 273 274 275 276 277 278 279 280 281 282 283 284 285 286 287 288 289 290 291 292 293 294 295 296 297 298 299 300 301 302 303 304 305 306 307 308 309 310 311 312 313 314 315 316 317 318 319 320 321 322 323 324 325 326 327 328 329 330 331 332 333 334 335 336 337 338 339 340 341 342 343 344 345 346 347 348 349 350 351 352 353 354 355 356 357 358 359 360 361 362 363 364 365 366 367 368 369 370 371 372 373 374 375 376 377 378 379 380 381 382 383 384 385 386 387 388 389 390 391 392 393 394 395 396 397 398 399 400 401 402 403 404 405 406 407 408 409 410 411 412 413 414 415 416 417 418 419 420 421 422 423 424 425 426 427 428 429 430 431 432 433 434 435 436 437 438 439 440 441 442 443 444 445 446 447 448 449 450 451 452 453 454 455 456 457 458 459 460 461 462 463 464 465 466 467 468 469 470 471 472 473 474 475 476 477 478 479 480 481 482 483 484 485 486 487 488 489 490 491 492 493 494 495 496 497 498 499 500 501 502 503 504 505 506 507 508 509 510 511 512 513 514 515 516 517 518 519 520 521 522 523 524 525 526 527 528 529 530 531 532 533 534 535 536 537 538 539 540 541 542 543 544 545 546 547 548 549 550 551 552 553 554 555 556 557 558 559 560 561 562 563 564 565 566 567 568 569 570 571 572 573 574 575 576 577 578 579 580 581 582 583 584 585 586 587 588 589 590 591 592 593 594 595 596 597 598 599 600 601 602 603 604 605 606 607 608 609 610 611 612 613 614 615 616 617 618 619 620 621 622 623 624 625 626 627 628 629 630 631 632 633 634 635 636 637 638 639 640 641 642 643 644 645 646 647 648 649 650 651 652 653 654 655 656 657 658 659 660 661 662 663 664 665 666 667 668 669 670 671 672 673 674 675 676 677 678 679 680 681 682 683 684 685 686 687 688 689 690 691 692 693 694 695 696 697 698 699 700 701 702 703 704 705 706 707 708 709 710 711 712 713 714 715 716 717 718 719 720 721 722 723 724 725 726 727 728 729 730 731 732 733 734 735 736 737 738 739 740 741 742 743 744 745 746 747 748 749 750 751 752 753 754 755 756 757 758 759 760 761 762 763 764 765 766 767 768 769 770 771 772 773 774 775 776 777 778 779 780 781 782 783 784 785 786 787 788 789 790 791 792 793 794 795 796 797 798 799 800 801 802 803 804 805 806 807 808 809 810 811 812 813 814 815 816 817 818 819 820 821 822 823 824 825 826 827 828 829 830 831 832 833 834 835 836 837 838 839 840 841 842 843 844 845 846 847 848 849 850 851 852 853 854 855 856 857 858 859 860 861 862 863 864 865 866 867 868 869 870 871 872 873 874 875 876 877 878 879 880 881 882 883 884 885 886 887 888 889 890 891 892 893 894 895 896 897 898 899 900 901 902 903 904 905 906 907 908 909 910 911 912 913 914 915 916 917 918 919 920 921 922 923 924 925 926 927 928 929 930 931 932 933 934 935 936 937 938 939 940 941 942 943 944 945 946 947 948 949 950 951 952 953 954 955 956 957 958 959 960 961 962 963 964 965 966 967 968 969 970 971 972 973 974 975 976 977 978 979 980 981 982 983 984 985 986 987 988 989 990 991 992 993 994 995 996 997 998 999 1000 1001 1002 1003 1004 1005 1006 1007 1008 1009 1010 1011 1012 1013 1014 1015 1016 1017 1018 1019 1020 1021 1022 1023 1024 1025 1026 1027 1028 1029 1030 1031 1032 1033 1034 1035 1036 1037 1038 1039 1040 1041 1042 1043 1044